

Book Chapter 12
Monad Comprehensions:
A Versatile Representation for Queries

Torsten Grust
University of Konstanz

In line with the major theme of the book, we perceive query translation and transformation as a *functional programming activity*.

Functional composition through **catamorphisms** will be the predominant way of forming complex queries. Referential transparency is the key to *transformational programming (manipulating catamorphisms)* and *equational reasoning*.

Core Language Types

$t ::= \mathbb{N} \mid \mathbb{B} \mid \mathbb{S} \mid \dots$	atomic (numeric, boolean, string, ...)
v	variables ($\alpha, \beta, \gamma, \dots$)
$t \rightarrow t$	functions
$t \times t$	pairs
$[t] \mid \{\!\! t \!\!\}$ $\{t\}$	list (bag, set) type constructor

Constructing Collections

Remember that we are growing this language for a specific purpose: to represent database query languages. So, where a typical functional language would offer *lists* only, the core supports the *collection types* *bags* (multi-sets) and *sets* as well. Again, this is a means to properly reflect the type system of the query language: SQL primarily operates on bags, while OQL includes clauses that operate on all three collection types.

Starting from an empty collection ($[], \{\},$ or $\{\}$), we can *insert* elements one by one using constructor \uparrow to construct a more complex collection value.

To aid compact notation, we define the *insertion constructor* \uparrow as overloaded, i.e. the type of its second argument determines its behaviour. Let $x :: \alpha$. Then:

$$x \uparrow xs = \begin{cases} [x] \# xs & \text{if } xs :: [\alpha] \\ \{x\} \uplus xs & \text{if } xs :: \{\alpha\} \\ \{x\} \cup xs & \text{if } xs :: \{\alpha\} \\ \text{type error} & \text{otherwise} \end{cases}$$

$\#$ denotes list concatenation, while \uplus is bag union respecting multiplicity of elements. Insertion of *duplicates* is respected if \uparrow constructs lists or bags. Set insertion $\uparrow :: \alpha \times \{\alpha\} \rightarrow \{\alpha\}$ disregards both order and duplicates, i.e. the constructor is *left-commutative* $y \uparrow x \uparrow xs = x \uparrow y \uparrow xs$ and/or *left-idempotent* $x \uparrow x \uparrow xs = x \uparrow xs$.

The expressive power of these **spine transformers** is sufficient to embrace almost all computations expressible by current database query languages. We adopt them as *the* basic query building block.

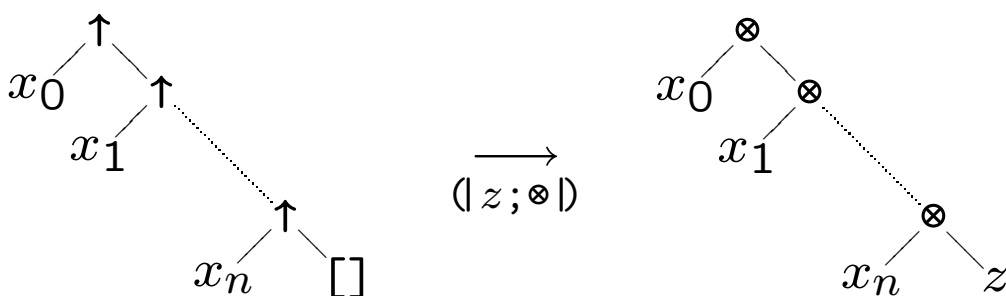
Catamorphism

Let us undertake a generalisation step.

Given a collection $[a]$ (or $\{a\}$, $\{a\}$) and values $z :: \beta$, $\otimes :: a \times \beta \rightarrow \beta$ we define the overloaded mix-fix operator (\Downarrow) as

$$\begin{aligned}
 (\Downarrow z; \otimes) &:: \beta \times (a \times \beta \rightarrow \beta) \rightarrow [a] \rightarrow \beta \\
 (\Downarrow z; \otimes) xs &\equiv \text{case } xs \text{ of } [] \quad \rightarrow z \\
 &\quad | x \uparrow xs' \rightarrow x \otimes ((\Downarrow z; \otimes) xs')
 \end{aligned}$$

Pictorially, $(\Downarrow z; \otimes)$ is the spine transformer



and we can immediately see that we could have defined $\text{maximum} \equiv (\Downarrow -\infty; \max)$. When applied to lists, the operator (\Downarrow) is known as **foldr** or **reduce**, in functional programming.

We are specially interested in cases where the starting value z used in `foldr f z 1` is a **zero** of the associative function f . Together they form a **monoid** (FEGARAS), for example:

$$0 + x = x + 0 = x$$

$$1 * x = x * 1 = x$$

$$\text{max} x - \infty = \text{max} - \infty x = x$$

We can give an algebraic account of the nature of (\downarrow) . Observe that $(\downarrow z; \otimes)$ is a solution to the equations below which effectively say that the unknown h is a *homomorphism* from $\text{monoid}([\uparrow], \uparrow)$ to $\text{monoid}(z, \otimes)$:

$$\begin{aligned} h [\uparrow] &= z \\ h (x \uparrow xs) &= x \otimes h xs \end{aligned}$$

It can be shown that $(\downarrow z; \otimes)$ is the *unique* solution to these equations, completely determined by z and \otimes (LAMB-68). Homomorphisms of initial algebras have been dubbed *catamorphisms* (MFP-91).

Caveat: operator \otimes needs to have the same algebraic properties as \uparrow : associativity, left-commutativity (if $\uparrow :: \alpha \times \{\alpha\} \rightarrow \{\alpha\}$ or $\uparrow :: \alpha \times \{\alpha\} \rightarrow \{\alpha\}$), or left-idempotence (if $\uparrow :: \alpha \times \{\alpha\} \rightarrow \{\alpha\}$).

Catamorphism examples

Catamorphisms are a versatile tool. A number of useful collection processing functions turn out to be catamorphisms:

```
maximum  ≡ (|-∞;max|)
minimum  ≡ (|+∞;min|)
    or    ≡ (|false;v|)
    and   ≡ (|true;∧|)
  xs ⊕ ys ≡ (|ys;↑|) xs
    first ≡ (|0;fst|)
list_map f ≡ (|[];λ(x,xs) → (f x) ↑ xs|)
  flatten ≡ (|[];⊕|)
```

Note that infix operator \oplus is overloaded and behaves like $\#$, $\#$, or \cup depending on the type of its arguments. As given, `list_map` is well-defined on lists only. The same is true for function `first`: `fst` is neither left-commutative nor left-idempotent, an expression of the fact that there is no notion of a first element in a bag or set.

Catamorphism Fusion

A query translator and optimizer based on the core language we have defined so far would more closely resemble a *program transformation system* than a traditional query optimizer. Catamorphisms represent this restricted form of computation and in our case, simplicity enables optimisation.

Reconsider `list_map`. We can turn this function into a generic `map` catamorphism if we make its implicit use of the list constructors `[]` and `↑ :: α × [α] → [α]` explicit and thus define

$$\text{map } n \ c \ f \equiv (\!| n; \lambda(x, xs) \rightarrow c (f \ x, xs) \!|)$$

Now, `list_map f` \equiv `map [] (↑) f`,

`set_map f` \equiv `map {} (↑) f`, and

`bag_map f` \equiv `map {} |> (↑) f`.

Apart from this generalisation, factoring the constructors out of a catamorphism opens up an important optimisation opportunity: we can “reach inside” a catamorphism and influence the constructor replacement it performs.

Catamorphism Fusion Law

This is all we need to formulate a simple yet effective *catamorphism fusion law*. Let `cata` denote any catamorphism with constructors factored out as above; then

$$(\lambda z; \otimes) \cdot \text{cata } n \ c = \text{cata } z \ \otimes$$

Note that while the left hand side walks the spine twice, the righthand side computes the same result in a single spine traversal. With catamorphisms being the basic program building blocks, a typical program form will be catamorphism compositions. These composition chains can be shortened and simplified using laws. The two-step catamorphism chain below decides if there is *any* element in the input satisfying `p`. Catamorphism fusion merges the steps and yields a general-purpose existential quantifier `exists p`:

$$\text{exists } p \quad \equiv \quad \text{or} \cdot \text{map } \{\} \uparrow p \quad = \quad \text{map } \text{false} \vee p$$

Monad Comprehension

We have seen that catamorphisms represent a form of computation restrictive enough to enable mechanical program optimisations, yet expressive enough to provide a useful target for query translation.

However, we need to make sure that query translation actually yields nothing but compositions of catamorphisms. This is what we turn to now.

To achieve this goal, we grow our language once more to include the expressions of the *monad comprehension calculus* (WADLER-92, WONG-94). Its semantics can be explained in terms of catamorphisms, thus completing the desired query translation framework:

SQL as Comprehension

For example, here is how we can define `bag_map f` and `flatten`:

$$\begin{aligned} \text{bag_map } f \text{ } xs &\equiv \{ \{ f \ x \mid x \leftarrow xs \} \} \\ \text{flatten } xss &\equiv \{ x \mid xs \leftarrow xss, x \leftarrow xs \} \end{aligned}$$

SQL and OQL queries, like the following *semi-join* between relations r and s , may now be understood as yet more syntactic sugar (we will encounter many more examples in the sequel):

```
select a
  from r, s
  where p(joinpredicatelike)v1.x == v2.y
v1 ← r, v2 ← s, p
```

$$\equiv \{ v_1.a \mid \dots \}$$

NOTE (PG) Daplex is actually much closer to comprehensions, and it also allows integer ranges as generators.

Monad Operations (Wadler)

Comprehension syntax can be sensibly defined for any type constructor $[[\alpha]]$ with operations `mmap`, `zero`, `unit`, `join` obeying the laws of a *monad with zero* which—for our collection constructors—are as follows:

```
    join · unit    = id
  join · mmap unit = id
    join · join    = join · mmap join
    join · zero    = zero
  join · mmap zero = zero
```

Lists, bags, and sets are easily verified to be monad instances. Monads are a remarkably general concept that has been widely used by the functional programming community to study, among others, I/O, stateful computation, and exception handling (PEYTON-JONES-01). More importantly, though, we can exercise a large number of query transformations and optimisation exclusively in comprehension syntax.

Translation Scheme using Wadler Identities for Monad comprehension semantics

$$\begin{aligned}
\llbracket e \mid \rrbracket &\equiv \text{unit } e \\
\llbracket e \mid v \leftarrow e' \rrbracket :: \llbracket \alpha \rrbracket &\equiv \text{mmap } (\lambda v \rightarrow e) \ e' \\
\llbracket e \mid v \leftarrow e' \rrbracket :: [\alpha] &\equiv \text{mmap id } ([e \mid v \leftarrow e']) \\
\llbracket e \mid v \leftarrow e' \rrbracket :: \{\!\! \{ \alpha \}\!\! \} &\equiv \text{mmap id } (\{\!\! \{ e \mid v \leftarrow e' \}\!\! \}) \\
\llbracket e \mid v \leftarrow e' \rrbracket :: \{ \alpha \} &\equiv \text{mmap id } (\{ e \mid v \leftarrow e' \}) \\
\llbracket e \mid e' \rrbracket :: \mathbb{B} &\equiv \text{case } e' \text{ of true } \rightarrow \text{unit } e \mid \\
&\text{false } \rightarrow \text{zero} \\
\llbracket e \mid qs, qs' \rrbracket &\equiv \text{join } (\llbracket e \mid qs' \rrbracket \mid qs) \\
\text{zero} &\equiv \llbracket \rrbracket \\
\text{unit } e &\equiv \llbracket e \rrbracket \\
\text{mmap} &\equiv \text{map } \llbracket \rrbracket (\uparrow) \\
\text{join} &\equiv (\llbracket \rrbracket ; \oplus)
\end{aligned}$$

The scheme is applicable to bag and set comprehensions as well (simply consistently replace all occurrences of \llbracket, \rrbracket by $[,]$ or $\{\!, \!\}$ or $\{, \}$, respectively).

Syntactic Sugar for CataMorphisms

Monad comprehensions provide quite powerful syntactic sugar and will save us from juggling with complex catamorphism chains. Consider, for example, the translation of `filter p` (which evaluates predicate p against the elements of the argument list):

```
filter p xs
  ≡ [x | x ← xs, p x]
  = join ([ [x | p x] | x ← xs ])
  = (join · mmap (λx → [x | p x])) xs
  = map [] ⊕ (λx → [x | p x]) xs
  = map [] ⊕ (λx → case p x of true →
[x] | false → []) xs
```

Unravelling Deeply Nested Queries

A nested user-level query may be mapped rather straightforwardly into a nested comprehension (see the example query at the end of the last section). However, deriving anything but a nested-loops execution plan from a deeply nested query is a hard task. We are really better off trying *unnest* a nested query before we process it further.

The monad comprehension calculus provides particularly efficient yet simple hooks to attack this problem:

- Different types of query nesting lead to similar nested forms of monad comprehensions. Rather than maintaining and identifying a number of special nesting cases — (KIM-82,GAWO-87) on classifying nested SQL queries— we can concentrate on unnesting the relatively few comprehension forms.

Comprehension Normalisation Laws

Much of the unnesting work can, once more, be achieved by application of a small number of syntactic rewriting laws, the *normalisation rules* below. The rules form a *confluent* and *terminating* set of rewriting rules. As before, you can obtain the specific variants through a consistent replacement of \llbracket, \rrbracket_n by $[,]$ or $\{\}, \}$ or $\{, \}$, respectively:

$$\begin{aligned} \llbracket e \mid qs, v \leftarrow \llbracket \rrbracket_2, qs' \rrbracket_1 &= \llbracket \rrbracket_1 \\ \llbracket e \mid qs, v \leftarrow \llbracket e' \rrbracket_2, qs' \rrbracket_1 &= \llbracket e[e'/v] \mid qs, qs'[e'/v] \rrbracket_1 \\ \llbracket e \mid qs, v \leftarrow \llbracket e' \mid qs'' \rrbracket_2, qs' \rrbracket_1 &= \llbracket e[e'/v] \mid qs, qs'', qs'[e'/v] \rrbracket_1 \\ \{e \mid qs, \text{or } \llbracket e' \mid qs'' \rrbracket, qs'\} &= \{e \mid qs, qs'', e', qs'\} \end{aligned}$$

(Expression $e[e'/v]$ denotes e with all free occurrences of v replaced by e' .)

Use of Normalisation Laws

Unnesting disentangles queries and makes operands of formerly inner queries accessible in the outer enclosing comprehension. This, in turn, provides new possibilities for further rewritings and optimisations. Comprehension syntax provides a rather poor variety of syntactical forms, but in the early stages of query translation this is more of a virtue than a shortcoming. Monad comprehensions extract and emphasize the *structural* gist of a query rather than stressing the diversity of query constructs.

Steenhagen, Apers, and Blanken (1994) analysed a class of SQL-like queries which exhibit correlated nesting in the *where*-clause, more specifically:

$$\begin{array}{l} \text{select distinct } f\ x \\ \quad \text{from } xs \text{ as } x \\ \quad \text{where } p\ x\ z \end{array} \quad \text{with } z = \left(\begin{array}{l} \text{select } g\ x\ y \\ \quad \text{from } ys \text{ as } y \\ \quad \text{where } q\ x\ y \end{array} \right)$$

Verifying a Flatten Transform'n

The question is, can queries of this class be rewritten into *flat join queries* of the form

```
select distinct  $f\ x$ 
      from  $xs$  as  $x$ ,  $ys$  as  $y$ 
  where  $q\ x\ y$ 
      and  $p'\ x\ (g\ x\ y)$ 
```

Queries for which such a replacement predicate p' cannot be found have to be processed either (a) using a nested-loops strategy, or (b) by grouping.

The monad comprehension normalisation rules provide an elegant proof of this transformation:

```
select distinct  $f\ x$ 
      from  $xs$  as  $x$ 
  where  $p\ x\ z$ 
= {  $f\ x \mid x \leftarrow xs, p\ x\ z$  }
= {  $f\ x \mid x \leftarrow xs, \text{or } \llbracket p'\ x\ v \mid v \leftarrow z \rrbracket$  }
= {  $f\ x \mid x \leftarrow xs, v \leftarrow z, p'\ x\ v$  }
= {  $f\ x \mid x \leftarrow xs, v \leftarrow \llbracket g\ x\ y \mid y \leftarrow ys, q\ x\ y \rrbracket, p'\ x\ v$  }
= {  $f\ x \mid x \leftarrow xs, y \leftarrow ys, q\ x\ y, p'\ x\ (g\ x\ y)$  }
```

More Unnesting

Monad comprehension normalisation readily unnests queries of Kim's *type J*, i.e. SQL queries of the form

$$Q \equiv \begin{array}{l} \text{select distinct } f\ x \\ \text{from } xs \text{ as } x \\ \text{where } p\ x \text{ in (select } g\ y \\ \text{from } ys \text{ as } y \\ \text{where } q\ x\ y) \end{array}$$

Note that predicate q refers to query variable x so that the outer and nested query blocks are correlated. (The SQL predicate `in` is translated into an existential quantifier.) The derivation of the normal form for this query effectively yields Kim's *canonical 2-relation query*:

$$\begin{aligned} Q &= \{f\ x \mid x \leftarrow xs, \text{ or } \llbracket p\ x = v \mid v \leftarrow \llbracket g\ y \mid y \leftarrow ys, q\ x\ y \rrbracket \rrbracket\} \\ &= \{f\ x \mid x \leftarrow xs, \text{ or } \llbracket p\ x = g\ y \mid y \leftarrow ys, q\ x\ y \rrbracket\} \\ &= \{f\ x \mid x \leftarrow xs, y \leftarrow ys, q\ x\ y, p\ x = g\ y\} \end{aligned}$$

We can see that Kim's *type J* unnesting is sound only if the outer query block is evaluated in the set monad. No such restriction, though, is necessary for the inner block—an immediate consequence of the well-definedness conditions for monad comprehension coercion.

Parallelizing Group-By Queries

The database back-ends of decision support or data mining systems frequently face SQL queries of the following general type (termed *group queries* in (CHRO-97)):

$$Q f g a xs \quad \equiv \quad \begin{array}{l} \text{select } f x, a (g x) \\ \text{from } xs \text{ as } x \\ \text{group by } f x \end{array}$$

Group queries extract a particular dimension or feature—described by function f —from given base data xs and then pair each data point $f x$ in this dimension with aggregated data $a (g x)$ associated with that point; a may be instantiated by any of the SQL aggregate functions, e.g. `sum` or `max`.

Example Group-By Query(PG)

A specific example for computing the total sum of salaries for employees over each department would be:

SQL:

```
select dept(x), sum(salary(x))
from xs as x
where ...
group by dept
```

Daplex:

```
for each d in department
print(dept(d), total(over x in employee
                    such that dname(x)=dept(d)
                    of salary(x)));
```

Jython

```
[(dept(d), total({salary(x) for x in employee()
                  if dname(x) == dept(d)})
  for d in department())]
```

Group-by as Monad Comprehension

Here is query Q expressed in the monad comprehension calculus (the group by introduces nesting in the outer comprehension's head):

$$Q \ f \ g \ a \ xs \equiv \{(f \ x, (agg \ a) \ \& \ g \ y \mid y \leftarrow xs, f \ y = f \ x) \mid x \leftarrow xs\}$$

Helper function agg translates SQL aggregates into their implementing catamorphisms, e.g., $agg \ \text{sum} = \langle 0; + \rangle$ and $agg \ \text{max} = \text{maximum}$.

We are essentially stuck with the inherent nesting. Normalisation is of no use in this case (the query is in normal form already).

Parallelizing in Partitions

Chatziantoniou and Ross (1997) thus propose a three-step route to process this type of query:

1. *partition* input xs with respect to f .
2. Evaluate a simplified variant Q' of Q on each partition. $Q' g a ps \equiv \text{select } a (g x) \text{ from } ps \text{ as } x$
or, equivalently,
 $Q' g a ps \equiv (agg a) \{ g y \mid y \leftarrow ps \}$
3. Finally, merge the results obtained in step 2 to form the query response.

This strategy clearly shows its benefit in step 2: first, since we may execute Q' on the different partitions in parallel. Second, there is a chance of processing the Q' in main memory should the partitions ps fit. In (CHRO-97), classical relational algebra is the target language for

the translation of group queries. Reasoning in the monad comprehension calculus can significantly simplify the matter. We can construct a correctness proof for the strategy which is basically built from the *unfolding of definitions* and *normalisation steps*. Let us proceed by filling the two gaps (partitioning and iteration) that relational algebra has left open.

First, partitioning the base data collection xs with respect to a function f is expressible as follows (note that we require type β to allow equality tests):

$$\begin{aligned} \text{partition} &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow \{(\beta, [\alpha])\} \\ \text{partition } f \text{ } xs &\equiv \{(f \ x, [y \mid y \leftarrow xs, f \ x = f \ y]) \mid x \leftarrow xs\} \end{aligned}$$

which builds a set of disjoint partitions such that all elements inside one partition agree on feature f with the latter attached to its associated partition. We have, for example,

$$\begin{aligned} \text{partition odd } [1 \dots 5] &= \\ \{(\text{true}, [1, 3, 5]), (\text{false}, [2, 4])\} \end{aligned}$$

Iteration and Partition

Second, recall that iteration forms a core building block of our functional language by means of `map`; `map f` also adequately encodes parallel application of `f` to the elements of its argument. (HILL-94)

We can compose the phases and express the complete parallel grouping plan as

$$(\text{map } \{\} (\uparrow) (\lambda(z, ps) \rightarrow (z, Q' g a ps))) \cdot \text{partition } f) xs$$

We can now derive a purely calculational proof of the correctness of the parallel grouping idea through a sequence of simple rewriting steps: unfold the definitions of Q' , `partition`, and `map`, then apply monad comprehension normalisation to finally obtain $Q f g a xs$, the original group query:

$$(\text{map } \{\} (\uparrow) (\lambda(z, ps) \rightarrow (z, Q' g a ps))) \cdot \text{partition } f) xs$$

$$\begin{aligned}
& \stackrel{=}{=} (\text{map } \{\} (\uparrow) (\lambda(z, ps) \rightarrow (z, Q' g a ps)) (\text{partition } (\cdot))) \\
& \stackrel{=}{=} \text{partition } (\text{map } \{\} (\uparrow) (\lambda(z, ps) \rightarrow (z, Q' g a ps)) \\
& \quad \{(f x, \{y \mid y \leftarrow xs, f x = f y\}) \mid x \leftarrow xs\}) \\
& \stackrel{=}{=} Q' (\text{map } \{\} (\uparrow) (\lambda(z, ps) \rightarrow (z, (agg a) \{g y' \mid y' \leftarrow ps\} \\
& \quad \{(f x, \{y \mid y \leftarrow xs, f x = f y\}) \mid x \leftarrow xs\} \\
& \stackrel{=}{=} \text{map } \{(\lambda(z, ps) \rightarrow (z, (agg a) \{g y' \mid y' \leftarrow ps\}) v \mid \\
& \quad v \leftarrow \{(f x, \{y \mid y \leftarrow xs, f x = f y\}) \mid x \leftarrow xs\}) \\
& \stackrel{=}{=} \{(f x, (agg a) \{g y' \mid y' \leftarrow \{y \mid y \leftarrow xs, f x = f y\}) \\
& \stackrel{=}{=} \{(f x, (agg a) \{g y \mid y \leftarrow xs, f x = f y\}) \mid x \leftarrow xs\} \\
& \stackrel{=}{=} Q f g a xs
\end{aligned}$$

Conclusion

We have seen that a monad comprehension $\llbracket f \ x \mid x \leftarrow xs \rrbracket$ can describe a variety of query constructs, e.g.

- parallel application of f to the elements of xs ,
- iteration for group-by,
- duplicate elimination,
- a quantifier ranging over xs

depending on which monad we are evaluating the comprehension in.

This uniformity has enabled us to spot useful and sometimes *unexpected dualities* between query constructs, e.g. the close connection

of the class of flat join queries and existential quantification.

The *terseness* of the calculus *reduces the size* of the rule sets necessary to express complex query rewrites, most notably normalisation.

We have found this purely functional representation of queries based on *catamorphisms and monads* to cover, simplify, and generalize many of the proposed views of classical database query languages as well as the more recent **XML languages** such as **XPath**.